Національний університет «Києво-Могилянська академія»

**ЗВІТ**

із лабораторної роботи №2

Викладач: к. н. Жежерун О. П.

Роботу виконав: Швачка Д. І.

План виконання роботи

1. Для роботи я обрав виконання операцій над об’єктом прямокутник, що був наданий мені на опрацювання в майбутній лабораторній. З метою закладання фундаменту була обрана саме ця фігура.
2. Програма перед початком аналізу розбиває текст на лексеми. У етапі підготовки вона зменшує регістр всіх символів окрім назв точок та прямокутників, що спрощує синтаксичний аналіз у подальшому. Також є можливість писати прості вирази у чисельні лексеми (наприклад, «2 + 7» воно замінить на «9»). Окрім цього є функціонал перетворення різних мір довжини до стандарту програми, що визначений у сантиметрах (наприклад, «2 м» воно замінить на «200», у той час як «1000 мм» воно замінить на «100»).
3. Граматика LL(1), що була побудована для синтаксичного аналізу:

<statement-list> ::= (<statement> ";")+

<statement> ::= <define-rectangle>

| <calculate-area>

| <calculate-perimeter>

| <build-diagonal>

| <move-by-rectangle>

| <move-to-rectangle>

| <resize-rectangle>

| <rotate-rectangle>

| <reflect-rectangle>

| <build-rectangle-with-ratio>

| <mark-intersection>

| <rename-point>

<define-rectangle> ::= "позначити" <rect>? <rectangle-name> (<coords-prefix>? <parameters>)?

<calculate-area> ::= "визначити площу" <rect>? <rectangle-name>?

<calculate-perimeter> ::= "визначити периметр" <rect>? <rectangle-name>?

<build-diagonal> ::= "побудувати діагональ" <rect>? <rectangle-name>?

<move-by-rectangle> ::= "здвинути" <rect>? <rectangle-name>? "на"? <vector>

<move-to-rectangle> ::= "перемістити" <rect>? <rectangle-name>? ("на" | "до" | "у" | "в")? <vector>

<resize-rectangle> ::= "змінити розмір" <rect>? <rectangle-name>? ("в" | "у")? <coefficient> ("раз")?

<rotate-rectangle> ::= "повернути" <rect>? <rectangle-name>? "на"? <angle> "з центром"? <center-point>

<reflect-rectangle> ::= "відзеркалити" <rect>? <rectangle-name>? "за"? <axis>

<build-rectangle-with-ratio> ::= "побудувати" <rect>? <rectangle-name> "на основі відношен" <ratio> <length> <coords-prefix>? ("точ")? <coordinate-pair>

<mark-intersection> ::= "позначити перетин" <rect>? <rectangle-name> ("та" | "з")? <rect>? <rectangle-name>?

<rename-point> ::= ("переіменувати" | "змінити назву") ("точ")? ("в" | "у")? <rect>? <rectangle-name>? ("точ")? <point-name> "на"? <point-name>

<rect> ::= "прямокут"

<coords-prefix> ::= "з координатами" | "в" | "у"

<coordinate-list> ::= <coordinate-pair> ("," <coordinate-pair>){3}

<coordinate-pair> ::= "(" <signed-number> "," <signed-number> ")"

<size-define> ::= <height-prefix>? <height> <width-prefix>? <width>

<parameters> ::= (<size-define> (("у точці" | "у" | "в")? <coordinate-pair>)?) | <coordinate-list>

<height-prefix> ::= "з висотою" | "висота" | "розміром"

<width-prefix> ::= "x" | "х" | "з шириною" | "ширина" | "на"

<vector> ::= "(" <signed-number> "," <signed-number> ")"

<coefficient> ::= <float>

<angle> ::= <number> ("°" | "'" | ("градус"))?

<center-point> ::= <point-name> | <coordinate-pair>

<axis> ::= "X" | "Y"

<ratio> ::= <number> (":" | "до") <number>

<point-name> ::= <latin-letter> <number>?

<rectangle-name> ::= <point-name>{4}

<width> ::= <number>

<height> ::= <number>

<length> ::= ("з діагоналлю" | "з довжиною")? <number>

1. Так як програма спеціалізується саме на побудові прямокутників, після функцій необов’язково писати слово «прямокутник» і тому подібне. Також програма запам’ятовує останній прямокутник з яким була операція, тому для наступних операцій над цим прямокутник не є обов’язковим написання його назви. Але якщо не писати назву прямокутника до позначення прямокутника, то, звісно, виникне помилка.

Окрім цього, програма має можливість коментування команди до “;”. Це може бути корисно для дебагу та просто приємно для користування. На жаль, нема можливості у БНФ зобразити правило «усі символи окрім», тому там воно описано умовно.

1. Приклади функцій на вході в програму:

позначити прямокутник ABCD;

ЗДВИНУТИ прямокутник НА (-5, -3);

визначити площу;

//змінити розмір у 2 рази;

визначити периметр;

//Побудувати ДіАгональ прямокутника;

позначити прямокутник ABGD з координатами (5,5), (5,10), (15,10), (15,5);

перемістити прямокутник у (20, 20);

позначити прямокутник SEXY;

здвинути прямокутник на (7, -5);

//позначити перетин ABCD;

здвинути прямокутник на (17, -15);

позначити прямокутник SL0N1K розміром 100 мм x 1 м у точці (5, -20);

1. Програма друкує дерево в консоль. Приклад:

define\_rectangle

rectangle\_name

point\_name

"A"

point\_name

"B"

point\_name

"C"

point\_name

"D "

move\_by\_rectangle

vector

signed\_number

"-5"

signed\_number

"-3"

calculate\_area

"визначити площу "

resize\_rectangle

coefficient

float

"2"

calculate\_perimeter

"визначити периметр "

build\_diagonal

"побудувати діагональ прямокутника "

define\_rectangle

rectangle\_name

point\_name

"A"

point\_name

"B"

point\_name

"G"

point\_name

"D "

coordinate\_list

coordinate\_pair

signed\_number

"5"

signed\_number

"5"

coordinate\_pair

signed\_number

"5"

signed\_number

"10"

coordinate\_pair

signed\_number

"15"

signed\_number

"10"

coordinate\_pair

signed\_number

"15"

signed\_number

"5"

move\_to\_rectangle

vector

signed\_number

"20"

signed\_number

"20"

1. Далі програма зберігає після вдалого аналізу результат у json файл. Після цього автоматично відпрацьовує команда запуску python скрипта, що цей результат зчитує та малює на полотні. Окрім простої побудови та операціями над прямокутниками, можна також розрахувати площу та периметр, що буде зазначатись у легенді полотна. Якщо над цим прямокутником надалі буде операція зміни розміру, то відповідні параметри оновляться автоматично.

**Код програми**

Пайтон скрипт для малювання:  
  
import json  
import matplotlib.pyplot as plt  
import matplotlib.patches as patches  
import random  
import sys  
from matplotlib.patches import Polygon  
  
def plot\_rectangles\_from\_json(json\_path):  
 with open(json\_path, 'r') as file:  
 rectangles = json.load(file)  
  
 fig, ax = plt.subplots()  
  
 colors = [(random.random(), random.random(), random.random()) for \_ in rectangles]  
 legend\_data = []  
  
 for idx, rect in enumerate(rectangles):  
 color = colors[idx]  
 points = [(point["x"], point["y"]) for point in rect["points"]]  
 is\_intersection = any(prop.get("IsIntersection", False) for prop in rect["properties"])  
  
 if is\_intersection:  
 polygon = Polygon(points, closed=True, linewidth=2, edgecolor=color, facecolor='none', linestyle="--", alpha=0.6)  
 else:  
 polygon = Polygon(points, closed=True, linewidth=2, edgecolor=color, facecolor=color, alpha=0.4)  
 ax.add\_patch(polygon)  
  
 for point in rect["points"]:  
 px, py = point["x"], point["y"]  
 name = point["name"]  
 ax.plot(px, py, 'o', color=color)  
 ax.text(px, py, f"{name}", ha="center", va="bottom", fontsize=11, color='black')  
  
 center\_x = sum(p[0] for p in points) / len(points)  
 center\_y = sum(p[1] for p in points) / len(points)  
 ax.text(center\_x, center\_y, rect["name"], ha="center", va="center", fontsize=14, color='black')  
  
 diagonal = next((prop["Diagonal"] for prop in rect["properties"] if "Diagonal" in prop), None)  
 if diagonal:  
 ax.plot([points[0][0], points[2][0]], [points[0][1], points[2][1]], color=color, linestyle="--")  
  
 if rect["properties"]:  
 properties\_text = ', '.join(  
 f"{list(prop.keys())[0]}: {round(list(prop.values())[0], 2)}" for prop in rect["properties"]  
 )  
 legend\_data.append((f"{rect['name']} - {properties\_text}", color))  
  
 legend\_handles = [patches.Patch(color=col, label=text) for text, col in legend\_data]  
 ax.legend(handles=legend\_handles, title="Rectangle Properties", loc="center left", bbox\_to\_anchor=(1, 0.5))  
  
 ax.set\_aspect('equal', adjustable='box')  
 plt.xlabel("X")  
 plt.ylabel("Y")  
 plt.title("Shvachka Denys Systemne Lab 2")  
 plt.show()  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 json\_path = sys.argv[1]  
 plot\_rectangles\_from\_json(json\_path)

Лексер:  
use regex::Regex;  
use std::collections::HashMap;  
use std::iter::Peekable;  
use std::str::Chars;  
use lazy\_static::lazy\_static;  
  
  
lazy\_static! {  
 static ref KEYWORDS: Vec<&*'static* str> = vec![  
 "позначити", "прямокутник", "побудувати", "визначити", "площу",  
 "периметр", "перемістити", "змінити", "розмір", "повернути",  
 "відзеркалити", "перетин", "точка", "висота", "ширина",  
 "діагональ", "довжина", "координати", "центр"  
 ];  
  
 static ref MEASUREMENT\_UNITS: HashMap<&*'static* str, f64> = {  
 let mut m = HashMap::*new*();  
 m.insert("m", 100.0);  
 m.insert("м", 100.0);  
 m.insert("метр", 100.0);  
 m.insert("cm", 1.0);  
 m.insert("см", 1.0);  
 m.insert("сантиметр", 1.0);  
 m.insert("mm", 0.1);  
 m.insert("мм", 0.1);  
 m.insert("міліметр", 0.1);  
 m  
 };  
 static ref POINT\_NAME\_REGEX: Regex = Regex::*new*(r"^[A-Z][0-9]\*$").unwrap();  
 static ref RECTANGLE\_NAME\_REGEX: Regex = Regex::*new*(r"^([A-Z][0-9]\*){4}$").unwrap();  
 static ref NUMBER\_REGEX: Regex = Regex::*new*(r"^[+-]?\d+(\.\d+)?$").unwrap();  
 static ref EXPRESSION\_REGEX: Regex = Regex::*new*(r"\(?\-?\d+(\.\d+)?[\+\-\\*\/]\d+(\.\d+)?\)?").unwrap();  
}  
  
pub struct Lexer {  
 text: String,  
}  
  
impl Lexer {  
 pub fn *new*(input: &str) -> Self {  
 Lexer {  
 text: input.to\_string(),  
 }  
 }  
 fn split\_with\_delimiters\_regex(&self) -> Vec<String> {  
 let mut result = Vec::*new*();  
 let mut current\_word = String::*new*();  
 let mut chars = self.text.chars().peekable();  
 let mut paren\_stack = Vec::*new*();  
 let mut is\_expression = false;  
  
 while let *Some*(c) = chars.next() {  
 match c {  
 '(' => {  
 if !current\_word.is\_empty() {  
 result.push(current\_word.clone());  
 current\_word.clear();  
 }  
  
 let mut look\_ahead = String::*new*();  
 let mut temp\_paren\_level = 1;  
 let mut found\_comma = false;  
 let mut peek\_chars = chars.clone();  
 let mut found\_close = false;  
  
 while let *Some*(next\_c) = peek\_chars.next() {  
 look\_ahead.push(next\_c);  
 match next\_c {  
 '(' => temp\_paren\_level += 1,  
 ')' => {  
 temp\_paren\_level -= 1;  
 if temp\_paren\_level == 0 {  
 found\_close = true;  
 break;  
 }  
 }  
 ',' => {  
 if temp\_paren\_level == 1 {  
 found\_comma = true;  
 break;  
 }  
 }  
 \_ => {}  
 }  
 }  
  
 if found\_comma || (found\_close && look\_ahead.split(',').count() == 2  
 && look\_ahead.chars().all(|c| c.is\_numeric() || c == ',' || c == ')' || c.is\_whitespace())) {  
 result.push(c.to\_string());  
 } else {  
 current\_word.push(c);  
 paren\_stack.push(true);  
 is\_expression = true;  
 }  
 }  
 ')' => {  
 if is\_expression {  
 current\_word.push(c);  
 if let *Some*(\_) = paren\_stack.pop() {  
 if paren\_stack.is\_empty() {  
 result.push(current\_word.clone());  
 current\_word.clear();  
 is\_expression = false;  
 }  
 }  
 } else {  
 if !current\_word.is\_empty() {  
 result.push(current\_word.clone());  
 current\_word.clear();  
 }  
 result.push(c.to\_string());  
 }  
 }  
 ',' | ';' | '\'' | '.' | '?' | '!' | ':' | '"' => {  
 if !is\_expression {  
 if !current\_word.is\_empty() {  
 result.push(current\_word.clone());  
 current\_word.clear();  
 }  
 result.push(c.to\_string());  
 } else {  
 current\_word.push(c);  
 }  
 }  
 ' ' => {  
 if !is\_expression {  
 if !current\_word.is\_empty() {  
 result.push(current\_word.clone());  
 current\_word.clear();  
 }  
 } else {  
 current\_word.push(c);  
 }  
 }  
 \_ => {  
 current\_word.push(c);  
 }  
 }  
 }  
  
 if !current\_word.is\_empty() {  
 result.push(current\_word);  
 }  
  
 result  
 }  
  
 pub fn process(&mut self) -> String {  
 let words: Vec<String> = self.split\_with\_delimiters\_regex();  
 let mut result = Vec::*new*();  
 let mut i = 0;  
  
 //println!("word: {:?}", words);  
 while i < words.len() {  
 let word = &words[i];  
  
 if word.trim().is\_empty() {  
 i += 1;  
 continue;  
 }  
 if RECTANGLE\_NAME\_REGEX.is\_match(word) {  
 result.push(word.clone());  
 } else if POINT\_NAME\_REGEX.is\_match(word) {  
 result.push(word.clone());  
 } else if let *Some*(processed) = self.process\_measurement(word, words.get(i + 1).map(|s| s.as\_str())) {  
 result.push(processed);  
 if words.get(i + 1).map\_or(false, |next| self.has\_unit(*Some*(next))) {  
 i += 1;  
 }  
 } else if let *Some*(expr) = self.try\_parse\_expression(word) {  
 result.push(expr);  
 } else {  
 result.push(word.to\_lowercase());  
 }  
 i += 1;  
 }  
  
 result.join(" ")  
 }  
  
 fn try\_parse\_expression(&self, word: &str) -> Option<String> {  
 if !word.contains(|c: char| ['+', '-', '\*', '/', '(', ')'].contains(&c)) {  
 return *None*;  
 }  
  
 if word.chars().count() == 1 && (word == "(" || word == ")") {  
 return *Some*(word.to\_string());  
 }  
  
 let mut evaluator = ExpressionEvaluator::*new*(word);  
 match evaluator.evaluate() {  
 *Ok*(result) => *Some*(result.to\_string()),  
 *Err*(\_) => *None*,  
 }  
 }  
  
 fn process\_measurement(&self, number: &str, next\_word: Option<&str>) -> Option<String> {  
 if !NUMBER\_REGEX.is\_match(number) {  
 return *None*;  
 }  
  
 let value = number.parse::<f64>().ok()?;  
  
 if let *Some*(unit) = next\_word {  
 for (unit\_name, coefficient) in MEASUREMENT\_UNITS.iter() {  
 if unit.starts\_with(unit\_name) {  
 return *Some*((value \* coefficient).to\_string());  
 }  
 }  
 }  
  
 *Some*(value.to\_string())  
 }  
  
 fn has\_unit(&self, word: Option<&str>) -> bool {  
 if let *Some*(w) = word {  
 for unit in MEASUREMENT\_UNITS.keys() {  
 if w.starts\_with(unit) {  
 return true;  
 }  
 }  
 }  
 false  
 }  
}  
#[derive(Debug, Clone)]  
enum Token {  
 *Number*(f64),  
 *Plus*,  
 *Minus*,  
 *Multiply*,  
 *Divide*,  
 *LeftParen*,  
 *RightParen*,  
}  
  
#[derive(Debug)]  
struct ExpressionEvaluator {  
 input: String,  
}  
  
impl ExpressionEvaluator {  
 fn *new*(input: &str) -> Self {  
 ExpressionEvaluator {  
 input: input.replace(" ", ""),  
 }  
 }  
  
 fn tokenize(&mut self) -> Result<Vec<Token>, String> {  
 let mut tokens = Vec::*new*();  
 let mut chars = self.input.chars().peekable();  
  
 while let *Some*(&c) = chars.peek() {  
 match c {  
 '0'..='9' | '.' => {  
 tokens.push(Token::*Number*(self.parse\_number(&mut chars)?));  
 }  
 '+' => {  
 chars.next();  
 tokens.push(Token::*Plus*);  
 }  
 '-' => {  
 chars.next();  
 if tokens.is\_empty() || matches!(tokens.last(), *Some*(Token::*LeftParen*)) {  
 if let *Some*(&next\_char) = chars.peek() {  
 if next\_char.is\_digit(10) {  
 let num = self.parse\_number(&mut chars)?;  
 tokens.push(Token::*Number*(-num));  
 continue;  
 }  
 }  
 }  
 tokens.push(Token::*Minus*);  
 }  
 '\*' => {  
 chars.next();  
 tokens.push(Token::*Multiply*);  
 }  
 '/' => {  
 chars.next();  
 tokens.push(Token::*Divide*);  
 }  
 '(' => {  
 chars.next();  
 tokens.push(Token::*LeftParen*);  
 }  
 ')' => {  
 chars.next();  
 tokens.push(Token::*RightParen*);  
 }  
 ' ' => {  
 chars.next();  
 }  
 \_ => return *Err*(format!("Неочікуваний символ: {}", c)),  
 }  
 }  
 *Ok*(tokens)  
 }  
  
 fn parse\_number(&self, chars: &mut Peekable<Chars>) -> Result<f64, String> {  
 let mut number\_str = String::*new*();  
  
 while let *Some*(&c) = chars.peek() {  
 if c.is\_digit(10) || c == '.' {  
 number\_str.push(c);  
 chars.next();  
 } else {  
 break;  
 }  
 }  
  
 number\_str.parse::<f64>()  
 .map\_err(|\_| "Parsing error.".to\_string())  
 }  
  
 fn evaluate(&mut self) -> Result<f64, String> {  
 let tokens = self.tokenize()?;  
 self.evaluate\_expression(&tokens, 0).map(|(result, \_)| result)  
 }  
  
 fn evaluate\_expression(&self, tokens: &[Token], pos: usize) -> Result<(f64, usize), String> {  
 let mut result = 0.0;  
 let mut current\_pos = pos;  
 let mut operation = Token::*Plus*;  
  
 while current\_pos < tokens.len() {  
 let (value, new\_pos) = match &tokens[current\_pos] {  
 Token::*Number*(n) => (\*n, current\_pos + 1),  
 Token::*LeftParen* => {  
 let (sub\_result, new\_pos) = self.evaluate\_expression(tokens, current\_pos + 1)?;  
 (sub\_result, new\_pos)  
 }  
 Token::*RightParen* => return *Ok*((result, current\_pos + 1)),  
 \_ => {  
 operation = tokens[current\_pos].clone();  
 current\_pos += 1;  
 continue;  
 }  
 };  
  
 result = match operation {  
 Token::*Plus* => result + value,  
 Token::*Minus* => result - value,  
 Token::*Multiply* => result \* value,  
 Token::*Divide* => {  
 if value == 0.0 {  
 return *Err*("Null division error.".to\_string());  
 }  
 result / value  
 }  
 \_ => result,  
 };  
  
 current\_pos = new\_pos;  
 }  
  
 *Ok*((result, current\_pos))  
 }  
}

mod lexer;  
  
use pest::iterators::{Pair, Pairs};  
use pest::Parser;  
use pest\_derive::Parser;  
use regex::Regex;  
use std::collections::{HashMap, HashSet};  
use std::error::Error;  
use std::fs::File;  
use std::path::PathBuf;  
use serde::{Deserialize, Serialize};  
use serde\_json::to\_writer;  
use crate::lexer::Lexer;  
use crate::ParseError::\*;  
use crate::RectangleProperties::{*Area*, *Diagonal*, *Perimeter*};  
  
#[derive(Parser)]  
#[grammar = "grammar.pest"]  
struct GrammarParser;  
  
#[derive(Debug)]  
pub enum ParseError {  
 *IncorrectName*(String),  
 *IncorrectValues*(String),  
 *IncorrectCoordinates*(String),  
 *IncorrectInput*(String),  
 *RectangleNotFound*(String),  
}  
#[derive(Debug, Clone, Serialize, Deserialize)]  
enum RectangleProperties {  
 *Perimeter*(f64),  
 *Area*(f64),  
 *Diagonal*(f64),  
 *IsIntersection*(bool),  
}  
#[derive(Debug, Clone, Serialize, Deserialize)]  
pub struct Point {  
 x: i32,  
 y: i32,  
 name: String,  
}  
#[derive(Debug, Clone, Serialize, Deserialize)]  
pub struct Rectangle {  
 w: i32,  
 h: i32,  
 points: Vec<Point>,  
 name: String,  
 properties: Vec<RectangleProperties>,  
}  
  
impl Rectangle {  
 pub fn *new*(x: i32, y: i32, w: i32, h: i32, name: &String) -> Result<Rectangle, ParseError> {  
 let point\_names = Self::*process\_name*(name)?;  
  
 let p1 = Point {  
 x,  
 y,  
 name: point\_names[0].clone(),  
 };  
 let p2 = Point {  
 x,  
 y: y + h,  
 name: point\_names[1].clone(),  
 };  
 let p3 = Point {  
 x: x + w,  
 y: y + h,  
 name: point\_names[2].clone(),  
 };  
 let p4 = Point {  
 x: x + w,  
 y,  
 name: point\_names[3].clone(),  
 };  
  
 let points = vec![p1, p2, p3, p4];  
  
 *Ok*(Rectangle {  
 w,  
 h,  
 points,  
 name: name.clone(),  
 properties: vec![]  
 })  
 }  
  
 pub fn *new\_from\_coords*(name: &String, coords: &[(i32, i32)]) -> Result<Rectangle, ParseError> {  
 let point\_names = Self::*process\_name*(name)?;  
  
 if coords.len() != 4 {  
 return *Err*(*IncorrectInput*(  
 "Exactly 4 coordinates are required to define a rectangle.".to\_string(),  
 ));  
 }  
  
 let p1 = Point { x: coords[0].0, y: coords[0].1, name: point\_names[0].clone() };  
 let p2 = Point { x: coords[1].0, y: coords[1].1, name: point\_names[1].clone() };  
 let p3 = Point { x: coords[2].0, y: coords[2].1, name: point\_names[2].clone() };  
 let p4 = Point { x: coords[3].0, y: coords[3].1, name: point\_names[3].clone() };  
  
 let side1 = ((p1.x - p2.x).pow(2) + (p1.y - p2.y).pow(2)) as f64;  
 let side2 = ((p2.x - p3.x).pow(2) + (p2.y - p3.y).pow(2)) as f64;  
 let side3 = ((p3.x - p4.x).pow(2) + (p3.y - p4.y).pow(2)) as f64;  
 let side4 = ((p4.x - p1.x).pow(2) + (p4.y - p1.y).pow(2)) as f64;  
  
 if (side1 - side3).abs() > 0.001 || (side2 - side4).abs() > 0.001 {  
 return *Err*(*IncorrectInput*(  
 "Provided coordinates do not form a rectangle.".to\_string(),  
 ));  
 }  
  
 let w = (p2.x - p1.x).abs().max((p4.x - p1.x).abs());  
 let h = (p3.y - p1.y).abs().max((p4.y - p1.y).abs());  
  
 *Ok*(Rectangle {  
 w,  
 h,  
 points: vec![p1, p2, p3, p4],  
 name: name.clone(),  
 properties: vec![]  
 })  
 }  
  
  
 pub fn *new\_with\_ratio*(name: &String, ratio: (u32, u32), length: u32, x: i32, y: i32) -> Result<Rectangle, ParseError> {  
 Self::*process\_name*(name)?;  
  
 if ratio.0 == 0 || ratio.1 == 0 {  
 return *Err*(*IncorrectInput*(  
 "Incorrect ratio.".to\_string()  
 ));  
 }  
  
 let total\_ratio = ratio.0 + ratio.1;  
 let unit\_length = length as f32 / total\_ratio as f32;  
 let w = (unit\_length \* ratio.0 as f32) as i32;  
 let h = (unit\_length \* ratio.1 as f32) as i32;  
  
 Rectangle::*new*(x, y, w, h, name)  
 }  
  
 fn *process\_name*(name: &String) -> Result<Vec<String>, ParseError> {  
 let re = Regex::*new*(r"[A-Z][0-9]\*").unwrap();  
 let point\_names: Vec<String> = re  
 .find\_iter(name)  
 .map(|mat| mat.as\_str().to\_string())  
 .collect();  
  
 if point\_names.len() != 4 {  
 return *Err*(*IncorrectName*(format!(  
 "Incorrect name format: {}. Must contain exactly 4 unique point names.",  
 name  
 )));  
 }  
  
 let unique\_names: HashSet<\_> = point\_names.iter().cloned().collect();  
 if unique\_names.len() != 4 {  
 return *Err*(*IncorrectName*(format!(  
 "Duplicate point names in: {}. Points must be unique.",  
 name  
 )));  
 }  
  
 *Ok*(point\_names)  
 }  
  
 pub fn has\_point(&self, name: &String) -> bool {  
 self.points.iter().any(|p| p.name == \*name)  
 }  
  
 pub fn area(&mut self) -> i32 {  
 self.properties.retain(|p| !matches!(p, *Area*(\_)));  
 let area = (self.w.abs() \* self.h.abs()) as f64;  
 self.properties.push(*Area*(area));  
 area as i32  
 }  
  
 pub fn perimeter(&mut self) -> i32 {  
 self.properties.retain(|p| !matches!(p, *Perimeter*(\_)));  
 let perimeter = 2.0 \* (self.w.abs() as f64 + self.h.abs() as f64);  
 self.properties.push(*Perimeter*(perimeter));  
 perimeter as i32  
 }  
  
 pub fn diagonal(&mut self) -> i32 {  
 self.properties.retain(|p| !matches!(p, *Diagonal*(\_)));  
 let diagonal = ((self.w.pow(2) + self.h.pow(2)) as f64).sqrt();  
 self.properties.push(*Diagonal*(diagonal));  
 diagonal as i32  
 }  
  
 fn update\_calculations(&mut self) {  
 if let *Some*(\_index) = self.properties.iter().position(|prop| matches!(prop, *Area*(\_))) {  
 self.area();  
 }  
 if let *Some*(\_index) = self.properties.iter().position(|prop| matches!(prop, *Perimeter*(\_))) {  
 self.perimeter();  
 }  
 if let *Some*(\_index) = self.properties.iter().position(|prop| matches!(prop, *Diagonal*(\_))) {  
 self.diagonal();  
 }  
 }  
  
 pub fn get\_position(&self) -> (i32, i32) {  
 (self.points[0].x, self.points[0].y)  
 }  
  
 pub fn move\_rectangle(&mut self, dx: i32, dy: i32) {  
 for point in &mut self.points {  
 point.x += dx;  
 point.y += dy;  
 }  
 }  
  
 pub fn move\_to(&mut self, x: i32, y: i32) {  
 self.points[0].x = x;  
 self.points[0].y = y;  
  
 self.points[1].x = x;  
 self.points[1].y = y + self.h;  
  
 self.points[2].x = x + self.w;  
 self.points[2].y = y + self.h;  
  
 self.points[3].x = x + self.w;  
 self.points[3].y = y;  
 }  
  
 pub fn resize\_rectangle(&mut self, factor: f32) {  
 self.w = (self.w as f32 \* factor) as i32;  
 self.h = (self.h as f32 \* factor) as i32;  
 let pos = self.get\_position();  
 for point in &mut self.points {  
 point.x = (pos.0 as f32 + (point.x - pos.0) as f32 \* factor) as i32;  
 point.y = (pos.1 as f32 + (point.y - pos.1) as f32 \* factor) as i32;  
 }  
 self.update\_calculations();  
 }  
  
 pub fn rotate\_rectangle(&mut self, angle: f32, center\_x: i32, center\_y: i32) {  
 let angle\_rad = angle.to\_radians();  
 for point in &mut self.points {  
 let translated\_x = (point.x - center\_x) as f32;  
 let translated\_y = (point.y - center\_y) as f32;  
 point.x = center\_x  
 + (translated\_x \* angle\_rad.cos() - translated\_y \* angle\_rad.sin()).round() as i32;  
 point.y = center\_y  
 + (translated\_x \* angle\_rad.sin() + translated\_y \* angle\_rad.cos()).round() as i32;  
 }  
 }  
  
 pub fn reflect\_rectangle(&mut self, axis: char) {  
 let pos = self.get\_position();  
 for point in &mut self.points {  
 match axis {  
 'X' => point.y = pos.1 - (point.y - pos.1),  
 'Y' => point.x = pos.0 - (point.x - pos.0),  
 \_ => {}  
 }  
 }  
 }  
  
 pub fn intersection(&self, other: &Rectangle) -> Option<Rectangle> {  
 let x1 = self.points[0].x.max(other.points[0].x);  
 let y1 = self.points[0].y.max(other.points[0].y);  
 let x2 = self.points[2].x.min(other.points[2].x);  
 let y2 = self.points[1].y.min(other.points[1].y);  
  
 if x1 < x2 && y1 < y2 {  
 let intersection\_name = format!(  
 "I{}{}",  
 self.name.chars().next().unwrap\_or('X'),  
 other.name.chars().next().unwrap\_or('Y')  
 );  
  
 let points = vec![  
 Point { x: x1, y: y1, name: format!("{}1", intersection\_name) },  
 Point { x: x1, y: y2, name: format!("{}2", intersection\_name) },  
 Point { x: x2, y: y2, name: format!("{}3", intersection\_name) },  
 Point { x: x2, y: y1, name: format!("{}4", intersection\_name) },  
 ];  
  
 let w = x2 - x1;  
 let h = y2 - y1;  
  
 *Some*(Rectangle {  
 w,  
 h,  
 points,  
 name: intersection\_name,  
 properties: vec![RectangleProperties::*IsIntersection*(true)],  
 })  
 } else {  
 *None* }  
 }  
  
 pub fn rename\_point(&mut self, old\_name: &str, new\_name: &str) -> Result<(), ParseError> {  
 let re = Regex::*new*(r"^[A-Z][0-9]\*$").unwrap();  
 if !re.is\_match(new\_name) {  
 return *Err*(*IncorrectName*(  
 format!("Incorrect new name format: {}.Must be a capital letter and optional digits.", new\_name)  
 ));  
 }  
  
 if self.points.iter().any(|p| p.name == new\_name) {  
 return *Err*(*IncorrectName*(  
 format!("The point name {} is already used in the rectangle.", new\_name)  
 ));  
 }  
  
 if let *Some*(point) = self.points.iter\_mut().find(|p| p.name == old\_name) {  
 point.name = new\_name.to\_string();  
 self.name = self.points.iter()  
 .map(|p| p.name.clone())  
 .collect::<Vec<\_>>()  
 .join("");  
 *Ok*(())  
 } else {  
 *Err*(*IncorrectName*(  
 format!("Point {} not found.", old\_name)  
 ))  
 }  
 }  
}  
  
pub fn parse(source: &str, path: &PathBuf) -> Result<(), ParseError> {  
 //println!("source: {:?}", source);  
 let processed = Lexer::*new*(source).process();  
 println!("processed: {:?}", processed);  
  
 let mut rectangles = HashMap::*new*();  
 let mut last\_rectangle: Option<String> = *None*;  
  
 let pairs = GrammarParser::*parse*(Rule::*statement\_list*, &processed)  
 .map\_err(|e| *IncorrectInput*(format!("Parse error: {}", e)))?;  
 //println!("Parsed pairs: {:?}", pairs);  
  
 fn get\_rectangle\_name(  
 pairs: &mut Pairs<Rule>,  
 last\_rectangle: &Option<String>,  
 ) -> Result<String, ParseError> {  
 match pairs.clone().next() {  
 *Some*(p) if p.as\_rule() == Rule::*rectangle\_name* => {  
 pairs.next();  
 *Ok*(p.as\_str().to\_string())  
 },  
 *Some*(\_) => last\_rectangle.clone().ok\_or\_else(|| {  
 *RectangleNotFound*("No rectangle specified.".to\_string())  
 }),  
 *None* => last\_rectangle.clone().ok\_or\_else(|| {  
 *RectangleNotFound*("No rectangle specified.".to\_string())  
 }),  
 }  
 }  
  
 fn parse\_coordinate\_pair(pair: Pair<Rule>) -> Result<(i32, i32), ParseError> {  
 if pair.as\_rule() != Rule::*coordinate\_pair* && pair.as\_rule() != Rule::*vector* {  
 return *Err*(*IncorrectInput*(  
 format!(  
 "Expected {:?}, but found a {:?}",  
 Rule::*coordinate\_pair*,  
 pair.as\_rule()  
 )  
 .to\_string(),  
 ));  
 }  
 let mut coords = pair.into\_inner();  
 let x = coords  
 .next()  
 .and\_then(|p| p.as\_str().parse::<i32>().ok())  
 .ok\_or\_else(|| {  
 *IncorrectInput*("Incorrect X coordinate format.".to\_string())  
 })?;  
 let y = coords  
 .next()  
 .and\_then(|p| p.as\_str().parse::<i32>().ok())  
 .ok\_or\_else(|| {  
 *IncorrectInput*("Incorrect Y coordinate format.".to\_string())  
 })?;  
 *Ok*((x, y))  
 }  
 fn parse\_coordinate\_list(coordinate\_list: Pair<Rule>) -> Result<Vec<(i32, i32)>, ParseError> {  
 if coordinate\_list.as\_rule() != Rule::*coordinate\_list* {  
 return *Err*(*IncorrectInput*(  
 format!(  
 "Expected {:?}, but was found a {:?}",  
 Rule::*coordinate\_list*,  
 coordinate\_list.as\_rule()  
 )  
 .to\_string(),  
 ));  
 }  
 let mut coordinates = Vec::*new*();  
 for pair in coordinate\_list.into\_inner() {  
 match pair.as\_rule() {  
 Rule::*coordinate\_pair* => {  
 coordinates.push(parse\_coordinate\_pair(pair)?);  
 }  
 \_ => {  
 return *Err*(*IncorrectInput*(  
 "Incorrect coordinate format.".to\_string(),  
 ))  
 }  
 }  
 }  
 *Ok*(coordinates)  
 }  
 display\_parse\_tree(pairs.clone(), 0);  
 for pair in pairs {  
 //println!("{:?}", pair.as\_rule());  
 match pair.as\_rule() {  
 Rule::*define\_rectangle* => {  
 let mut inner = pair.into\_inner();  
 let name = get\_rectangle\_name(&mut inner, &last\_rectangle)?;  
 if let *Some*(coords\_list) = inner.next() {  
 match coords\_list.as\_rule() {  
 Rule::*coordinate\_list* => {  
 let coords = parse\_coordinate\_list(coords\_list)?;  
 let rect = Rectangle::*new\_from\_coords*(&name, &coords)?;  
 rectangles.insert(name.clone(), rect);  
 last\_rectangle = *Some*(name.clone());  
 }  
 Rule::*size\_define* => {  
 let mut params\_inner = coords\_list.into\_inner();  
 let height = params\_inner  
 .find(|p| p.as\_rule() == Rule::*height*)  
 .and\_then(|p| p.into\_inner().next())  
 .and\_then(|p| p.as\_str().parse::<i32>().ok())  
 .ok\_or\_else(|| *IncorrectInput*("Incorrect height".to\_string()))?;  
  
 let width = params\_inner  
 .find(|p| p.as\_rule() == Rule::*width*)  
 .and\_then(|p| p.into\_inner().next())  
 .and\_then(|p| p.as\_str().parse::<i32>().ok())  
 .ok\_or\_else(|| *IncorrectInput*("Incorrect width".to\_string()))?;  
  
 let (x, y) = if let *Some*(coord\_pair) = inner.next() {  
 parse\_coordinate\_pair(coord\_pair)?  
 } else {  
 (0, 0)  
 };  
  
 let rect = Rectangle::*new*(x, y, width, height, &name)?;  
 rectangles.insert(name.clone(), rect);  
 last\_rectangle = *Some*(name.clone());  
 }  
 \_ => {},  
 }  
 } else {  
 let rect = Rectangle::*new*(0, 0, 10, 20, &name)?;  
 rectangles.insert(name.clone(), rect);  
 last\_rectangle = *Some*(name.clone());  
 }  
 }  
 Rule::*calculate\_area* => {  
 let name = get\_rectangle\_name(&mut pair.into\_inner(), &last\_rectangle)?;  
 let rect = rectangles  
 .get\_mut(&name)  
 .ok\_or(*RectangleNotFound*(format!("Rectangle {} not found", name.clone())))?;  
 last\_rectangle = *Some*(name.clone());  
 println!("Area of rectangle {}: {}", name, rect.area());  
 }  
 Rule::*calculate\_perimeter* => {  
 let name = get\_rectangle\_name(&mut pair.into\_inner(), &last\_rectangle)?;  
 let rect = rectangles  
 .get\_mut(&name)  
 .ok\_or(*RectangleNotFound*(format!("Rectangle {} not found", name.clone())))?;  
 last\_rectangle = *Some*(name.clone());  
 println!("Perimeter of rectangle {}: {}", name, rect.perimeter());  
 }  
 Rule::build\_diagonal => {  
 let name = get\_rectangle\_name(&mut pair.into\_inner(), &last\_rectangle)?;  
 let rect = rectangles  
 .get\_mut(&name)  
 .ok\_or(*RectangleNotFound*(format!("Rectangle {} not found", name.clone())))?;  
 last\_rectangle = *Some*(name.clone());  
 println!("Diagonal length of rectangle {}: {}", name, rect.diagonal());  
 }  
 Rule::move\_by\_rectangle => {  
 let mut inner = pair.into\_inner();  
  
 let name = get\_rectangle\_name(&mut inner, &last\_rectangle)?;  
 let vector = inner  
 .next()  
 .filter(|p| p.as\_rule() == Rule::*vector*)  
 .ok\_or\_else(|| *IncorrectInput*("The movement vector not found.".to\_string()))?;  
  
 let (dx, dy) = parse\_coordinate\_pair(vector)?;  
 let rect = rectangles  
 .get\_mut(&name)  
 .ok\_or(*RectangleNotFound*(format!("Rectangle {} not found", name.clone())))?;  
 last\_rectangle = *Some*(name.clone());  
 rect.move\_rectangle(dx, dy);  
 }  
 Rule::move\_to\_rectangle => {  
 let mut inner = pair.into\_inner();  
  
 let name = get\_rectangle\_name(&mut inner, &last\_rectangle)?;  
  
 let vector = inner  
 .next()  
 .filter(|p| p.as\_rule() == Rule::*vector*)  
 .ok\_or\_else(|| *IncorrectInput*("The movement vector not found.".to\_string()))?;  
  
 let (dx, dy) = parse\_coordinate\_pair(vector)?;  
 let rect = rectangles  
 .get\_mut(&name)  
 .ok\_or(*RectangleNotFound*(format!("Rectangle {} not found", name.clone())))?;  
 last\_rectangle = *Some*(name.clone());  
 rect.move\_to(dx, dy);  
 }  
 Rule::*resize\_rectangle* => {  
 let mut inner = pair.into\_inner();  
 let name = get\_rectangle\_name(&mut inner, &last\_rectangle)?;  
 let factor = inner  
 .next()  
 .filter(|p| p.as\_rule() == Rule::*coefficient*)  
 .and\_then(|p| p.as\_str().parse::<f32>().ok())  
 .ok\_or\_else(|| *IncorrectInput*("The coefficient factor not found.".to\_string()))?;  
  
 let rect = rectangles  
 .get\_mut(&name)  
 .ok\_or(*RectangleNotFound*(format!("Rectangle {} not found", name.clone())))?;  
 last\_rectangle = *Some*(name.clone());  
 rect.resize\_rectangle(factor);  
 }  
 Rule::*rotate\_rectangle* => {  
 let mut inner = pair.into\_inner();  
 let name = get\_rectangle\_name(&mut inner, &last\_rectangle)?;  
 let angle = inner  
 .next()  
 .and\_then(|p| p.into\_inner().next().unwrap().as\_str().parse::<f32>().ok())  
 .ok\_or\_else(|| *IncorrectInput*("The angle not found or incorrect.".to\_string()))?;  
  
 let center = inner  
 .next()  
 .filter(|p| p.as\_rule() == Rule::*center\_point*)  
 .ok\_or\_else(|| *IncorrectInput*("The rotate center not found.".to\_string()))?;  
  
 let center\_value = center.into\_inner().next().unwrap();  
  
 let rect = rectangles  
 .get\_mut(&name)  
 .ok\_or(*RectangleNotFound*(format!("Rectangle {} not found", name.clone())))?;  
  
 let (center\_x, center\_y) = match center\_value.as\_rule() {  
 Rule::*coordinate\_pair* => parse\_coordinate\_pair(center\_value)?,  
 Rule::*point\_name* => {  
 let point = rect.points.iter().find(|p| p.name.eq(center\_value.as\_str()));  
 match point {  
 *Some*(point) => (point.x, point.y),  
 *None* => return *Err*(*IncorrectInput*("Point not found.".to\_string()))  
 }  
 },  
 \_ => return *Err*(*IncorrectInput*("Неправильний формат центру обертання".to\_string()))  
 };  
  
 last\_rectangle = *Some*(name.clone());  
 rect.rotate\_rectangle(angle, center\_x, center\_y);  
  
 }  
 Rule::*reflect\_rectangle* => {  
 let mut inner = pair.into\_inner();  
 let name = get\_rectangle\_name(&mut inner, &last\_rectangle)?;  
  
 let axis = inner  
 .next()  
 .filter(|p| p.as\_rule() == Rule::*axis*)  
 .and\_then(|p| p.as\_str().chars().next())  
 .ok\_or\_else(|| *IncorrectInput*("The axis not found or incorrect.".to\_string()))?;  
  
 let rect = rectangles  
 .get\_mut(&name)  
 .ok\_or(*RectangleNotFound*(format!("Rectangle {} not found", name.clone())))?;  
 last\_rectangle = *Some*(name.clone());  
 rect.reflect\_rectangle(axis);  
 },  
 Rule::*build\_rectangle\_with\_ratio* => {  
 let mut inner = pair.into\_inner();  
 let name = get\_rectangle\_name(&mut inner, &last\_rectangle)?;  
  
 let mut ratio\_pair = inner  
 .next()  
 .ok\_or(*IncorrectInput*("The ratio not found or incorrect.".to\_string()))?.into\_inner();  
 let ratio = (ratio\_pair.next().unwrap().as\_str().parse::<u32>().unwrap(), ratio\_pair.next().unwrap().as\_str().parse::<u32>().unwrap());  
  
 let mut length\_pair = inner  
 .next()  
 .filter(|p| p.as\_rule() == Rule::*length*)  
 .ok\_or(*IncorrectInput*("The length not found or incorrect.".to\_string()))?.into\_inner();  
 let length = length\_pair.next().unwrap().as\_str().parse::<u32>().unwrap();  
  
 let coordinate\_pair = inner  
 .next()  
 .filter(|p| p.as\_rule() == Rule::*coordinate\_pair*)  
 .ok\_or\_else(|| *IncorrectInput*("The coordinate pair not found.".to\_string()))?;  
 let (dx, dy) = parse\_coordinate\_pair(coordinate\_pair)?;  
  
 let rect = Rectangle::*new\_with\_ratio*(&name, ratio, length, dx, dy)?;  
 last\_rectangle = *Some*(name.clone());  
 rectangles.insert(name.clone(), rect);  
 },  
 Rule::*mark\_intersection* => {  
 let mut inner = pair.into\_inner();  
  
 let name1 = match inner.next() {  
 *Some*(p) if p.as\_rule() == Rule::*rectangle\_name* => p.as\_str().to\_string(),  
 *Some*(p) => *Err*(*IncorrectInput*(format!("The rectangle name was expected, but found {:?}..", p.as\_rule()).to\_string()))?,  
 *None* => return *Err*(*IncorrectInput*("The rectangle name not found.".to\_string()))  
 };  
 let name2 = get\_rectangle\_name(&mut inner, &last\_rectangle)?;  
 let intersection = Rectangle::intersection(rectangles.get(&name1).unwrap(), rectangles.get(&name2).unwrap());  
 match intersection {  
 *None* => {  
 println!("Intersection of rectangles {} and {} not found", name1, name2);  
 }  
 *Some*(i) => {  
 rectangles.insert(i.name.clone(), i);  
 }  
 }  
 },  
 Rule::*rename\_point* => {  
 let mut inner = pair.into\_inner();  
 let name = get\_rectangle\_name(&mut inner, &last\_rectangle)?;  
  
 let mut rect = rectangles.remove(&name)  
 .ok\_or\_else(|| *IncorrectInput*(format!("Rectangle {} not found.", name)))?;  
  
 let p1 = inner  
 .next()  
 .filter(|p| p.as\_rule() == Rule::*point\_name*)  
 .ok\_or\_else(|| *IncorrectInput*("The point name was not found.".to\_string()))?.as\_str().to\_string();  
  
 if !rect.has\_point(&p1) {  
 return *Err*(*IncorrectInput*(format!("The point {} was not found in rectangle.", p1).to\_string()));  
 }  
  
 let p2 = inner  
 .next()  
 .filter(|p| p.as\_rule() == Rule::*point\_name*)  
 .ok\_or\_else(|| *IncorrectInput*("The point name was not found.".to\_string()))?.as\_str().to\_string();  
  
 match rect.rename\_point(&p1, &p2) {  
 *Ok*(()) => {  
 rectangles.insert(rect.name.clone(), rect);  
 println!("Successfully renamed the point.")  
 },  
 *Err*(e) => println!("Rename error: {:?}", e),  
 }  
 }  
  
 \_ => {}  
 }  
 }  
 let rectangles\_vec: Vec<Rectangle> = rectangles.values().cloned().collect();  
 save\_rectangles\_to\_json(&rectangles\_vec, path).unwrap();  
 *Ok*(())  
}  
fn save\_rectangles\_to\_json(rectangles: &Vec<Rectangle>, path: &PathBuf) -> Result<(), Box<dyn Error>> {  
 let file = File::*create*(path.join("rectangles.json"))?;  
 to\_writer(file, &rectangles)?;  
 *Ok*(())  
}  
fn display\_parse\_tree(pairs: Pairs<Rule>, depth: usize) {  
 for pair in pairs {  
 let indent = " ".repeat(depth);  
 println!("{} {:?}", indent, pair.as\_rule());  
  
 if pair.clone().into\_inner().count() > 0 {  
 display\_parse\_tree(pair.into\_inner(), depth + 2);  
 } else {  
 println!("{} {:?}", indent, pair.as\_str());  
 }  
 }  
}

мейн:  
use std::env;  
use std::error::Error;  
use std::path::PathBuf;  
use std::process::Command;  
use lab2::parse;  
  
fn main() -> Result<(), Box<dyn Error>> {  
 let source =  
 "позначити прямокутник ABCD;  
 ЗДВИНУТИ прямокутник НА (-5, -3);  
 визначити площу;  
 змінити розмір у 2 рази;  
 визначити периметр;  
 Побудувати ДіАгональ прямокутника;  
 позначити прямокутник ABGD з координатами (5,5), (5,10), (15,10), (15,5);  
 перемістити прямокутник у (20, 20);  
 позначити прямокутник SEXY;  
 здвинути прямокутник на (7, -5);  
 позначити перетин ABCD;  
 здвинути прямокутник на (17, -15);  
 позначити прямокутник SL0N1K розміром 100 мм x 1 м у точці (5, -20);  
  
 //переіменувати точку ABCD C на L;";  
  
 let project\_dir = env::var("CARGO\_MANIFEST\_DIR")  
 .expect("CARGO\_MANIFEST\_DIR environment variable not set");  
  
 let project\_path = PathBuf::*from*(project\_dir).join("src");  
 println!("Project directory: {:?}", project\_path);  
  
 match parse(source, &project\_path) {  
 *Ok*(\_) => {  
 Command::*new*("python")  
 .arg(project\_path.join("plot\_rectangles.py"))  
 .arg(project\_path.join("rectangles.json"))  
 .spawn()?  
 .wait()?;  
 },  
 *Err*(e) => println!("Error occurred: {:?}", e),  
 }  
 *Ok*(())  
}